Computer Chat

- How do we make computers talk?
- How are they interconnected?

**Internet Protocol (IP)**

- Datagram (packet) protocol
- Best-effort service
  - Loss
  - Reordering
  - Duplication
  - Delay
- Host-to-host delivery (we are not at 'application' level yet – more later)

**IP Address**

- 32-bit identifier (IPv4, IPv6=128 bits)
- Dotted-quad: 192.118.56.25 (readable)
- www.mkp.com -> 167.208.101.28
- Identifies a host **interface** (not a computer. could have multiple interfaces on a computer)

Like a Street Address

192.18.22.13 209.134.16.12

**Transport Protocols**

*Best-effort* is not sufficient!

- Add services on top of IP (higher level – abstractions)
- User Datagram Protocol (UDP)
  - Data checksum
  - Best-effort
- Transmission Control Protocol (TCP)
  - Data checksum
  - Reliable byte-stream delivery
  - Flow and congestion control

**Organize Protocols in Layers**
**Ports**

**Identifying the ultimate destination**
- IP addresses identify hosts
- Host has many applications
- Ports (16-bit identifier) 1-65,535 (about 2000 are reserved).

<table>
<thead>
<tr>
<th>Application</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>80</td>
</tr>
<tr>
<td>E-mail</td>
<td>25</td>
</tr>
<tr>
<td>Telnet</td>
<td>23</td>
</tr>
<tr>
<td>Echo</td>
<td>7</td>
</tr>
</tbody>
</table>

Like a Room Number

| IP address | 192.18.22.13 |

---

**Internet Phone Book**

- Domain Name Service (DNS)
  - Data base maps domain names to internet addresses

**Socket**

**How does one speak TCP/IP?**
- Sockets provides an interface to TCP/IP
- Generic interface for many protocols

Like a File Descriptor for a file

**TCP/IP Sockets:**

**Creates end point (and flavor)**

```c
int mySock = socket(family, type, protocol);
```
- TCP/IP-specific sockets

<table>
<thead>
<tr>
<th>Family</th>
<th>Type</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>PF_INET</td>
<td>IPPROTO_TCP</td>
</tr>
<tr>
<td>UDP</td>
<td></td>
<td>IPPROTO_UDP</td>
</tr>
</tbody>
</table>

- Socket reference
  - File (socket) descriptor in UNIX

**Type:** Semantics of transmission: e.g., is it reliable, best-effort, boundaries (packets, streams)

---

**Sockets**

- Identified by
  - protocol and
  - local/remote address/port
  - (both address and a port)
- Applications may refer to many sockets
- Sockets accessed by many applications

**Data Structure to Specifying Addresses**

```c
struct sockaddr
{
  unsigned short sa_family; /* Address family (e.g., AF_INET) */
  char sa_data[14]; /* BLOB */
  /* Protocol-specific address information */
};
```

```c
struct sockaddr_in
{
  unsigned short sin_family; /* Internet protocol (AF_INET) */
  short sin_port; /* Port (16-bits) */
  struct in_addr sin_addr; /* Internet address (32-bits) */
  char sin_zero[8]; /* Not used */
};
```

```c
struct in_addr
{
  unsigned long s_addr; /* Internet address (32-bits) */
};
```
struct sockaddr {
    unsigned short sa_family; /* Address family (e.g., AF_INET) */
    char sa_data[14];       /* Protocol-specific address information */
};

struct sockaddr_in /* TCP/IP structure form */ {
    unsigned short sin_family; /* Internet protocol (AF_INET) */
    unsigned short sin_port;  /* Port (16-bits) */
    struct in_addr sin_addr;  /* Internet address (32-bits) */
    char sin_zero[8];         /* Not used */
};

struct in_addr {
    unsigned long s_addr;    /* Internet address (32-bits) */
};

Note:

- **In Theory**: Protocol family to socket() (PF_INET) for internet family are different from the addressing scheme (AF_INET) – here it is 1-1 but does not need to be.
- **In Practice**: AF_XXXX and PF_XXXX constants are interchangeable. Values are the same AF_XXXX = PF_XXXX

Clients and Servers

**Server**: Waits until needed

**Client**: Initiates the connection

Client: Bob
"Hi. I'm Bob."

Server: Jane
"Hi, Bob. I'm Jane"

"Nice to meet you, Jane."

Two separate programs – works both on the same machine or on remote (e.g., nike, ajax)

TCP Client/Server Interaction

Server starts by getting ready to receive client connections...

TCP Client/Server Interaction

/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

TCP Client/Server Interaction

echoServAddr.sin_family = AF_INET; /* Internet address family */
if ((servSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection
TCP Client/Server Interaction

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

TCP Client/Server Interaction

/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
    DieWithError("listen() failed");

TCP Client/Server Interaction

for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);
    if ((clntSock = accept(servSock, (struct sockaddr *)&echoClntAddr, &clntLen)) < 0)
        DieWithError("accept() failed");

    echoServAddr.sin_family = AF_INET; /* Internet address family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
    echoServAddr.sin_port = htons(echoServPort); /* Server port */

    if (connect(clntSock, (struct sockaddr *)&servSockAddr, sizeof(servSockAddr)) < 0)
        DieWithError("connect() failed");

TCP Client/Server Interaction

/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

TCP Client/Server Interaction

for (;;) /* Run forever */
{
    clntLen = sizeof(echoClntAddr);
    if ((clntSock = accept(servSock, (struct sockaddr *)&echoClntAddr, &clntLen)) < 0)
        DieWithError("accept() failed");

    echoServAddr.sin_family = AF_INET; /* Internet address family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
    echoServAddr.sin_port = htons(echoServPort); /* Server port */

    if (connect(clntSock, (struct sockaddr *)&servSockAddr, sizeof(servSockAddr)) < 0)
        DieWithError("connect() failed");

TCP Client/Server Interaction

Later, a client decides to talk to the server...

Server is now blocked waiting for connection from a client

TCP Client/Server Interaction

/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

TCP Client/Server Interaction

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection
TCP Client/Server Interaction

```c
/* Determine input length */

/* Send the string to the server */
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
    DieWithError("send() sent a different number of bytes than expected");
```

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

```c
/* Receive message from client */
if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
    DieWithError("recv() failed");
```

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Tidbits
- Client knows server address and port
- No correlation between `send()` and `recv()`

```
Client
send("Hello Bob")
recv() -> "Hello"
recv() -> "Bob"
send("Hi")
send("Jane")
recv() -> "Hi Jane"
```

Closing a Connection
- `close()` used to delimit communication
- Analogous to EOF

```
Client
send(string)
while (not received entire string)
    recv(buffer)
send(buffer)
close(socket)
```

Server
recv(buffer)
while(client has not closed connection)
    send(buffer)
recv(buffer)
close(client socket)
TCP/IP Byte Transport

- TCP/IP protocols transports **bytes**
- Application protocol provides semantics
  - TCP does not examine or modify bytes

Here are some bytes. I don't know what they mean. I'll pass these to the app. It knows what to do.

---

Constructing Messages

... something to think about...

---

Application Protocol

- Encode information in **bytes**
- Sender and receiver must **agree on semantics**
- Data encoding
  - **Primitive types**: strings, integers, and etc.
  - **Composed types**: message with fields

---

Primitive Types

- **String**
  - Character encoding: ASCII, Unicode, UTF
  - Delimit: length vs. termination character

---

Primitive Types

- **Integer**
  - Strings of **character** encoded decimal digits
    
    
    49 55 57 57 55 55 48 10
    
    '1' '7' '9' '9' '8' '7' '0'

    - Advantage:
      - 1. Human readable
      - 2. Arbitrary size (in contrast to native integer format - fixed).
    
    - Disadvantage:
      - 1. Inefficient (space 1 byte (char) to represent 0-9, while only really need 4 bits).
      - 2. Arithmetic manipulation (must convert to integers)

---

Byte Ordering

- Computers orders bytes differently.
  - Big endian machines stores high order byte at lowest address (Big end first).
    - Example: The Network, Motorola power pc processor (early apple/mac systems in the 80s), Sun SPARC (Sun Java).
  - Little endian stores low order byte at lowest address (Little end first). Example: Intel x86, Window, and Intel Mac Systems

---

http://en.wikipedia.org/wiki/Endianness
Network Byte Order Functions

- **Network byte order** is in **Big-Endian** and used for multi-byte, binary data exchange.
- Hosts differ: either Big-Endian or Little Endian.
- `htonl()`, `htons()`, `ntohl()`, `ntohs()`:
  - Converts between **host byte order** and **network byte order**
  - `'h'` = host byte order
  - `'n'` = network byte order (big endian)
  - `'l'` = long (4 bytes), converts IP addresses
  - `'s'` = short (2 bytes), converts port numbers

- **Implementation**: If the byte order is already big-endian, then the `[hn]to[ns]s()` functions are no-ops.

Wikipedia Source

Other: Image/File Formats

Common file formats and their endian order are as follows:

- Adobe Photoshop -- Big-Endian
- BMP (Windows and OS/2 Bitmaps) -- Little Endian
- DXF (AutoCad) -- Variable
- GIF -- Little Endian
- IMG (GEM Raster) -- Big Endian
- JPEG -- Big Endian
- FLI (Autodesk Animator) -- Little Endian
- MacPaint -- Big Endian
- PCX (PC Paintbrush) -- Little Endian
- PostScript -- Not applicable (text)
- POV (Persistence of Vision ray-tracer) -- Not applicable (text)
- QT M (Quicktime Movies) -- Little Endian (on a Mac!)
- Microsoft RIFF (AVI & .AVI) -- Both
- Microsoft RTF (Rich Text Format) -- Little Endian
- SGI (Silicon Graphics) -- Big Endian
- Sun Raster -- Big Endian
- Targa -- Little Endian
- TIFF -- Both, endian identifier encoded into file
- VIPG (WordPerfect Graphics Metafile) -- Big Endian (on a PC!)
- XWD (X Window Dump) -- Both, endian identifier encoded into file

Message Composition

- **Message composed of fields**
  - Fixed-length fields
    - integer  short  short
  - Variable-length fields
    - `M i k e 1 2 \n`

Puzzle: What is the size of each structure/class

```
struct Alpha {
    int x;
    char ch1;
    int y;
    char ch2;
} a1;
```

- Assume int=4, char=1
- Compacted:
  - `A=4+1+4+1=10`
  - `B=4+4+1+1=10`

```
struct Beta {
    int x;
    int y;
    char ch_name[2];
} b1;
```

Puzzle: What is the size of each structure/class

```
struct Alpha {
    int x;
    char ch1;
    int y;
    char ch2;
} a1;
```

- Assume int=4, char=1
- Compacted:
  - `A=4+1+4+1=10`
  - `B=4+4+1+1=10`

```
printf("A = %d\n", (int)sizeof(a1)); = 16
printf("B = %d\n", (int)sizeof(b1)); = 12
```

- Compiler pads to align data members at multiple of word sizes
```c
#include <stdio.h>

struct Alpha {
    int x;
    char ch1;
    int y;
    char ch2;
} a1;

struct Beta {
    int x;
    int y;
    char ch_name[2];
} b1;

int main() {
    printf("A = %d\n", (int) sizeof(a1));
    printf("B = %d\n", (int) sizeof(b1));
}
```

```c
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */
#define RCVBUFSIZE 32 /* Size of receive buffer */

void DieWithError(char *errorMessage); /* Error handling function */

int main(int argc, char *argv[]) {
    int sock;                        /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    unsigned short echoServPort;     /* Echo server port */
    char *servIP;                    /* Server IP address (dotted quad) */
    char *echoString;                /* String to send to echo server */
    char echoBuffer[RCVBUFSIZE];     /* Buffer for echo string */
    unsigned int echoStringLen;      /* Length of string to echo */
    int bytesRcvd, totalBytesRcvd;   /* Bytes read in single recv() and total bytes read */

    if ((argc < 3) || (argc > 4))    /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [<Echo Port>]
", argv[0]);
        exit(1);
    }
    servIP = argv[1];             /* First arg: server IP address (dotted quad) */
    echoString = argv[2];         /* Second arg: string to echo */
    if (argc == 4)
    {
        echoServPort = atoi(argv[3]); /* Use given port, if any */
    } else
    {
        echoServPort = 7; /* 7 is the well-known port for the echo service */
    }
    /* Create a reliable, stream socket using TCP */
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    {
        DieWithError("socket() failed");
    }
    /* Construct the server address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr)); /* Zero out structure */
    echoServAddr.sin_family = AF_INET; /* Internet address family */
    echoServAddr.sin_port = htons(echoServPort); /* Server port */
    if (connect(sock, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr)) < 0)
    {
        DieWithError("connect() failed");
    }
    echoStringLen = strlen(echoString); /* Determine input length */
    if (argc < 3 || (argc > 4)) /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage: %s <Server IP> <Echo Word> <Echo Port>\n", argv[0]);
        exit(1);
    }

    /* receive the same string back from the server */
    totalBytesRcvd = 0;
    print("Received: "); /* Setup to print the echoed string */
    while (totalBytesRcvd < echoStringLen)
    {
        if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <= 0)
        {
            DieWithError("recv() failed or connection closed prematurely");
            totalBytesRcvd += bytesRcvd; /* Keep tally of total bytes */
            echoBuffer[bytesRcvd] = '\0'; /* Terminate the string */
            printf(echoBuffer); /* Print the echo buffer */
            if (bytesRcvd != echoStringLen)
            {
                printf("\n"); /* Print a final linefeed */
                close(sock);
                exit(0);
            }
        }
        print("\n"); /* Print a final linefeed */
    }
}
```

---

"Beware the bytes of padding"  
-- Julius Caesar, Shakespeare

**Architecture alignment restrictions**

- Compiler pads structs to accommodate

**Problem: Alignment restrictions vary**

**Solution:**

1. Rearrange struct members
2. Serialize struct by-member
TCPEchoServer.c

#include <stdio.h> /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h> /* for sockaddr_in and inet_ntoa() */
#include <string.h> /* for memchr() */
#include <unistd.h> /* for close() */

#define MAXPENDING 5 /* Maximum outstanding connection requests */

void DieWithError(char *errorMessage); /* Error handling function */
void HandleTCPClient(int clntSocket); /* TCP client handling function */

int main(int argc, char *argv[ ])
{
int servSock; /* Socket descriptor for server */
int clntSock; /* Socket descriptor for client */
struct sockaddr_in echoServAddr; /* Local address */
struct sockaddr_in echoClntAddr; /* Client address */
unsigned short echoServPort; /* Server port */
unsigned int clntLen; /* Length of client address data structure */

if (argc != 2) /* Test for correct number of arguments */
{
fprintf(stderr, "Usage:  %s <Server Port>
", argv[0]);
exit(1);
}

echoServPort = atoi(argv[1]); /* First arg: local port */

/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
DieWithError("socket() failed");

/* Construct local address structure */
memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
echoServAddr.sin_family = AF_INET;                /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);      /* Local port */

/* Bind to the local address */
if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
DieWithError("bind() failed");

/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
DieWithError("listen() failed");

for (;;) /* Run forever */
{
 handlers TCPClient(servSock);
 /* Set the size of the in-out parameter */
 clntLen = sizeof(echoClntAddr);

 /* Wait for a client to connect */
 if (clntSock = accept(servSock, (struct sockaddr*) &echoClntAddr, &clntLen) < 0)
 DieWithError("accept() failed");
 /* clntSock is connected to a client! */
 printf("Handling client %s
", inet_ntoa(echoClntAddr.sin_addr));

 how does the server know which client it is?
 • clnt.sin_addr_s addr contains the client’s IP address
 • clnt.sin_port contains the client’s port number
 • accept() is blocking (What does that mean?)
 • Why does accept need to return a new descriptor?
Dealing with blocking calls

- How do we deal with blocking calls?
  - `accept()`: until a connection comes in
  - `connect()`: until the connection is established
  - `recv()`: until a packet (of data) is received
  - `send()`: until data is pushed into socket’s buffer
    - Q: why not until received?

- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Why Multitasking?

- Previously server could only server one client at a time (iteratively).
- For computational intensive tasks how can we server multiple clients?
  - Multitasking allows servers to farm out work to other processes or threads each executing independently (possibly as a copy of the original server).

Dealing w/ blocking (cont’d)

- Options:
  - Implement multi-tasking:
    - create multi-process or multi-threaded code
  - turn off the blocking feature:
    - e.g., using the `fcntl` file-descriptor control function.
  - use the `select()` function call (on your own…)

(1) Server runs forever and listens for connections at a specific port and repeatedly accepts incoming connections from clients

(2) When a client connects (after `accept()`) it creates a new process to handle that connection via `fork()`

(3) Fork creates a new child process and copies socket descriptors.

(4)
(4) The child now deals with the Client and it does not listen on the servSock any more so it closes that connection. The Server only needs to listen for other new Clients so it closes clntSock.

#include <TCPEchoServer.h> /* TCP echo server includes */
#include <sys/socket.h>     /* for accept() */
int main(int argc, char *argv[]) {  
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    unsigned short echoServPort;    /* Server port */
    pid_t processID;                /* Process ID from fork() */
    unsigned int childProcCount = 0; /* Number of child processes */
    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port>
", argv[0]);
        exit(1);
    }
    echoServPort = atoi(argv[1]);  /* First arg:  local port */
    servSock = CreateTCPServerSocket(echoServPort);
    for (;;) /* Run forever */
    {
        clntSock = AcceptTCPConnection(servSock);  /* Fork child process and report any errors */
        if ((processID = fork()) < 0)
            DieWithError("fork() failed");
        else if (processID == 0)  /* If this is the child process */
            { close(servSock);   /* Child closes parent socket */
                HandleTCPClient(clntSock);  /* Handle TCP client connection */
                exit(0);           /* Child process terminates */
            }
        printf("with child process: %d\n", (int) processID);
        close(clntSock);       /* Parent closes child socket descriptor */
        childProcCount++;      /* Increment number of outstanding child processes */
        while (childProcCount) /* Clean up all zombies */
        {
            processID = waitpid((pid_t) -1, NULL, WNOHANG);  /* Non-blocking wait */
            if (processID < 0)  /* waitpid() error? */
                DieWithError("waitpid() failed");
            else if (processID == 0)  /* No zombie to wait on */
                break;
            else
                childProcCount--;  /* Cleaned up after a child */
        }
    }
}

#define MAXPENDING 5 /* Maximum outstanding connection requests */
void DieWithError(char *errorMessage);  /* Error handling function */
int CreateTCPServerSocket(unsigned short port)
{  
    int sock;                        /* Socket to create */
    struct sockaddr_in echoServAddr; /* Local address */
    /* Create socket for incoming connections */
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");
    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(port);              /* Local port */
    /* Bind to the local address */
    if (bind(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");
    /* Construct listening socket */
    if (listen(sock, MAXPENDING) < 0)
        DieWithError("listen() failed");
    /* Mark the socket so it will listen for incoming connections */
    if (listen(sock, MAXPENDING) < 0)
        DieWithError("listen() failed");
    return sock;
}
Practice Quiz

- What is the correct ordering of calls for an echo socket server? [send(), bind(), accept(), socket(), listen(), and recv()].
- What does htons() do?
- What is the name of the application that installs new software?
- What virtual machine have you installed on your computer?